

Algoritmi e strutture dati

Koci Erik

May 18, 2021

1 Complessità algoritmi

1. **Costo:** si riferisce al costo di un singolo algoritmo
2. **Complessità:** si riferisce a più risoluzioni di un algoritmo

il costo di un blocco **if-then-else** è $O(\max\{f(n), g(n), h(n)\})$ cioè **O(1)**.

1.1 Ordini di grandezza

1. $\Theta(f(n))$ se cresce tanto quanto f
2. $O(f(n))$ se la crescita è minore o uguale a f
3. $\Omega(f(n))$ se la crescita è maggiore o uguale a f

1.2 Esercizi

- $1325n^2 + 12n + 1 = \theta(n^3)$ FALSO
- $76n^3 == (n^3)$ VERO
- $n^2 \log n = O(n^2)$ FALSO
- $3^N = O(2^N)$ FALSO
- $1^n = O(2^{\frac{n}{2}})$ FALSO
- $2^N + 100 = O(2^N)$ VERO

- $n = O(n \log n)$ VERO
- $n^2 = (n \log n)$ FALSO
- $\log(n^2) = \Theta(\log n)$ VERO
- $(n + 1)/2 = \Theta(n)$ VERO
- $\frac{(n+1)*n}{2} = \Theta(n^2)$ VERO

1.3 Analisi casi

1. Caso pessimo

$$T_{\text{WORST}}(n) = \max T(I)$$

2. Caso ottimo

$$T_{\text{BEST}}(n) = \min T(I)$$

3. Caso medio

$$T_{\text{AVG}}(n) = \sum T(I)P(I)$$

1.4 Algoritmi ordinamento

Selection sort: scorre tutti gli elementi degli array e si cerca il valore più piccolo scambiando i due valori. Il costo è lineare con il numero di elementi da considerare:

$$T(n) = \Theta(n^2)$$

il costo è $\Theta(n^2)$ perchè è presente una funzione min che ogni volta controlla se il numero è il minore. Le chiamate a **min** contribuiscono a n^2 mentre il resto combacia con n cioè $n^2 + n$; n viene assorbito.

Ricerca binaria (ricorsiva): per utilizzare questo algoritmo devo avere un array ordinato. Cerco il valore andando a verificare sempre nella metà dove mi aspetto che sia presente.

$$T(n) = 1 \text{ se } n = 0$$

$$T(n) = T(n/2) + 1$$

equazione di ricorrenza: ci aiuta a calcolare il costo analizzando una singola ricorsione.

1.5 metodo dell'iterazione:

consiste nello sviluppare l'equazione di ricorrenza, per intuirne la soluzione.

$$T(n) = c_1 + c_2 * \log(n) = \Theta(\log(n))$$

E' presente il **logaritmo** perchè ogni volta devo **dimezzare** il tutto in base al numero di elementi. c_1 perchè devo eseguire le istruzioni la prima volta.

dimostrare che $T(n) = O(n)$

$$T(n = 1) \quad n == 1$$

$$T(T([n/2]) + n) \quad n > 1$$

1.6 Metodo della sostituzione:

consiste facendo una dimostrazione per induzione. quindi parto dal valore base che è n (esempio 1) e dimostriamo che vale anche per un n più grande.

caso base:

$$T(1) = 1 \leq cn$$

induzione:

$$\begin{aligned} T(n) &= T([n/2]) + n \\ &\leq c[n/2] + n \quad (\text{ipotesi induttiva}) \\ &\leq cn/2 + n = \frac{cn + 2n}{2} = (c/2 + 1)n \leq cn \end{aligned}$$

1.7 Master Theorem:

Si consideri la seguente equazione di ricorrenza:

$$\begin{aligned}T(n) &= d \text{ se } n = 1 \\T(n) &= aT(n/b) + cn^\beta \text{ se } n > 1\end{aligned}$$

e sia:

$$\alpha = \frac{\log(a)}{\log(b)}$$

a numero di chiamate ricorsive

b mi dice come partiziono il mio input

questi due valori mi danno α .

β mi dice l'esponente che avevo.

L'equazione di ricorrenza ha la seguente soluzione:

1. $T(n) = \Theta(n^\alpha)$ se $\alpha > \beta$
2. $T(n) = \Theta(n^\alpha * \log(n))$ se $\alpha = \beta$
3. $T(n) = \Theta(n^\beta)$ se $\alpha < \beta$

Il teorema fondamentale **non** si può applicare ad algoritmi ricorsivi che non effettuano **partizioni bilanciate**.

ad esempio non può essere applicato nella risoluzione di fibonacci ricorsivo.

Esempio:

$$\begin{aligned}T(n) &= 1 \text{ } n \leq 1 \\T(n1) + T(n2) + 1 &\text{ } n > 2\end{aligned}$$

Se le **partizioni sono bilanciate** conviene utilizzare il **Master Theorem**.

partizione bilanciate: quando facciamo chiamate ricorsive prendo il mio input suddividendolo in parti n/b . Fibonacci non è bilanciato perché abbiamo due chiamate ricorsive diverse.

L'analisi ammortizzata: studia il costo medio di una sequenza di operazioni.

Sia $T(n, k)$ il tempo totale richiesto da un algoritmo, nel caso pessimo, per effettuare k operazioni su istanze di lunghezza n . Definiamo il **costo ammortizzato** su una sequenza di k operazioni come:

$$T_{\alpha}(n) = \frac{T(n, k)}{k}$$

1.8 Algoritmi di visita degli alberi

Esistono due tipologie di visita:

- In profondità (pre-ordine, in-ordine, post-ordine)
- In ampiezza

Nella visita **pre-ordine** si parte visitando il nodo della radice per poi passare a visitare tutto il nodo sinistro, risalendo poi andando verso destra.

Nella visita **in-ordine** si parte a visitare dal ramo più in basso a sinistra risalendo per poi andare verso destra.

Nella visita **post-ordine** vengono prima visitati i nodi più in profondità partendo sempre da sinistra verso destra per poi risalire.

Nella visita per **ampiezza** si analizza l'albero a livelli, partendo dalla radice.

1.9 Alberi AVL

Un albero *AVL* è un albero di ricerca (quasi) bilanciato. Questo albero supporta le operazioni di *insert()*, *delete()*, *search()* con costo $O(\log n)$ nel caso pessimo.

1.9.1 Fattore di bilanciamento

Il fattore di bilanciamento $\beta(v)$ di un nodo v è dato dalla differenza tra l'altezza del sottoalbero sinistro e del sottoalbero destro di v :

$$\beta(v) = \text{altezza}(v.\text{left}) - \text{altezza}(v.\text{right})$$

1.9.2 Bilanciamento in altezza

Un albero si dice **bilanciato in altezza** se le altezze dei sottoalberi sinistro e destro di ogni nodo differiscono al più di uno.

$$\beta \leq 1$$

Definizione: un albero *AVL* è un *ABR* bilanciato in altezza.

1.9.3 Inserimento e rimozione

Inserimenti e rimozioni richiedono di essere modificati per mantenere il bilanciamento dell'albero.

L'operazione fondamentale per ribilanciare l'albero è la **rotazione semplice**.

1.9.4 Rotazione a sinistra

Per effettuare questa rotazione prendo il nodo problematico e effettuo una rotazione scambiandolo con il successivo ed il nodo scambiato diventerà figlio destro mentre il figlio del nodo precedente diventerà figlio sinistro.

1.10 Alberi 2-3

Un albero 2-3 è un albero in cui:

- Tutti i percorsi radice-foglia hanno la stessa lunghezza
- Le foglie contengono le chiavi (e i dati da memorizzare) e sono ordinate da sinistra verso destra in ordine di chiave crescente.
- Ogni nodo interno (non foglia) v ha 2 o 3 figli e mantiene due informazioni
 - $S[v]$, **chiave massima** nel sottoalbero sinistro (2 o 3 figli)
 - $M[v]$, **chiave massima** nel sottoalbero centrale (3 figli)
- Distribuzione dei valori k delle chiavi nei sottoalberi:
 - Sinistro $k \leq S[v]$
 - Centro $S[v] < k \leq M[v]$
 - Destro $k > M[v]$

1.11 Tabelle Hash

Le **tabelle hash** hanno una implementazione basata su una chiave k e array. Per ottenere la chiave sono presenti diverse tecniche di calcolo.

Ricapitolando, per realizzare una tabella hash efficiente abbiamo bisogno di:

- Un vettore
- Una funzione hash calcolabile velocemente e che garantisca una buona distribuzione delle chiavi nel vettore
- Un meccanismo per gestire le collisioni

1.11.1 Problema delle collisioni

Una funzione hash h si dice **perfetta** se è iniettiva:

$$\forall u, v \in U : u \neq v \rightarrow h(u) \neq h(v)$$

Se le collisioni sono inevitabili, è necessario trovare un metodo che le minimizzi, distribuendo **uniformemente** le chiavi negli indici della tabella hash.

1.11.2 Funzioni hash

E' necessario fare una premessa; nelle funzioni hash è **sempre possibile** trasformare una chiave complessa in un numero, (conversione in binario).

1.11.3 Metodo dell'estrazione

Le caratteristiche di questo metodo sono:

- Usa solo una parte della chiave
- Si seleziona una sottosequenza di p bit, con $m = 2^p$
- Solitamente dalle posizioni centrali

esempio: Verranno prese le cifre centrali 101000

$$\text{bin}(\text{"beer"}) = 000010\ 000101\ 000101\ 010010$$

- **Vantaggi:** molto veloce da calcolare
- **Svantaggi** rischio collisioni più alto di altri metodi

1.11.4 Metodo della divisione

Basata sul resto della divisione per m :

- **Vantaggio:** molto veloce
- **Svantaggio:** Suscettibile a specifici valori di m . Per risolvere questo problema bisogna scegliere m come numero primo non troppo vicino ad una potenza di 2.

Esempio:

$$m = 12, k = 100 \rightarrow h(k) = 4$$

1.11.5 Metodo della moltiplicazione

Basato sulla moltiplicazione e il resto del numero

1. Sia A una costante, $0 < A < 1$
2. Moltiplichiamo k per A e prendiamo la parte frazionaria
3. Moltiplichiamo quest'ultima per m e prendiamo la parte intera

Esempi:

$$m =, k = 3, A = 0.8 \rightarrow h(k) = 2$$

$$m = 1000, k = 123, A \approx 0.6180339887... \rightarrow h(k) = 18$$

- **Svantaggi:** lento (più lento del metodo di divisione)
- **Vantaggi** Il valore di m non è critico
- **Come scegliere A?** $A \approx (\sqrt{5} - 1)/2 = 0.61803... \text{ (Knuth)}$

1.11.6 Metodo della codifica algebrica

Metodo utilizzando dal compilatore java basato su espressioni algebriche:

$$h(k) = (k_n x^n + k_{n-1} x^{n-1} + \dots + k_1 x + k_0) \bmod m$$

$$k = k_n k_{n-1} \dots k_1 k_0$$

Dove $k_0, k_1 \dots$ possono essere, ad esempio, i bit della **codifica binaria** di k , oppure i **codici ascii** dei singoli caratteri di k .

x è un valore **costante**.

- **Vantaggi:** dipende da tutti i bit/caratteri della chiave
- **Svantaggi:** n addizioni e $n * (n - 1)/2$ prodotti

1.11.7 Problema delle collisioni

Attraverso questi metodi elencati precedentemente siamo riusciti a ridurre il numero di collisioni, ma senza eliminarle.

Per risolvere questo problema la **complessità computazionale potrebbe aumentare a n** , possono essere utilizzate le seguenti tecniche:

1. **Concatenamento**
2. **Indirizzamento aperto**

1.11.8 concatenamento

Nella tecnica di **concatenamento** gli elementi con lo stesso valore hash h vengono memorizzati in una lista concatenata (linked list).

Il **fattore di carico** è dato dal rapporto tra numero di elementi memorizzati e dimensioni della tabella.

La **complessità** del concatenamento è la seguente:

- insert: $\Theta(1)$
- search: $\Theta(n)$
- delete: $\Theta(n)$

1.11.9 indirizzamento aperto

L'idea è quella di memorizzare tutte le chiavi nella tabella stessa, ed ogni slot contiene una chiave oppure *null*.

Inserimento: se lo slot prescelto è utilizzato, si cerca uno slot alternativo.

Ricerca: si cerca nello slot prescelto, e poi negli slot alternativi fino a quando non si trova la chiave oppure *null*.

Vengono utilizzati diversi algoritmi di indirizzamento, per esempio i seguenti:

Ispezione lineare

Il primo elemento determina l'intera sequenza. In questo modo si ottengono **lunghe sottosequenze**.

$$h(k, i) = (h'(k) + i)$$

Ispezione quadratica:

L'ispezione iniziale è in $h'(k)$, mentre le successive hanno un offset che dipende da una **funzione quadratica nel numero di ispezione**.

$$h'(k) + c_1i + c_2i^2$$

Doppio hashing:

Formato da **due funzioni ausiliari** di cui la prima h_1 fornisce la prima ispezione, mentre h_2 fornisce l'offset delle successive ispezioni.

$$h(k, i) = (h_1(k) + ih_2(k))$$

1.11.10 Conclusioni hash table

Usare funzioni hash $h(k)$ che producano valori il più possibile uniformemente distribuiti è molto importante perchè altrimenti potremmo arrivare ad una complessità computazionale pari a $O(n)$.

Problemi con hashing:

- Scarsa locality of reference (cache miss)
- In base all'implementazione è in genere difficile ottenere le chiavi in ordine
- Sebbene il costo medio per operazione sia basso, la singola operazione può risultare molto costosa, ad esempio se occorre ridimensionare la tabella e redistribuire le chiavi.

2 Scelta degli algoritmi

A seconda delle operazioni da eseguire è necessario adattare diverse tecniche di implementazione di un algoritmo.

2.0.1 Implementazione su un vettore ordinato

Questo tipo di ricerca ha un costo computazionale basso nel caso in cui volessimo **ricercare degli elementi**. Costi computazionali:

- Ricerca $O(\log n)$
- Inserimento $O(n)$
- Eliminazione $O(n)$

2.0.2 Implementazione su liste concatenate non ordinate

Questa implementazione converrebbe utilizzarla nel caso in cui volessimo **aggiungere o eliminare degli elementi**. Costi computazionali:

- Ricerca $O(n)$
- Inserimento $O(1)$
- Eliminazione $O(n)$

2.0.3 Implementazione alberi ABR

Implementazione basata su alberi binari. Costi computazionali:

- Ricerca $O(h)$
- Inserimento $O(h)$
- Eliminazione $O(h)$

2.0.4 Implementazione alberi AVL

Implementazione basata su alberi binari a altezza equivalente. Costi computazionali:

- Ricerca $O(\log n)$
- Inserimento $O(\log n)$
- Eliminazione $O(\log n)$

2.0.5 Implementazione alberi 2-3

Implementazione basata su alberi binari ordinati con chiavi massime. Costi computazionali:

- Ricerca $O(\log n)$
- Inserimento $O(\log n)$
- Eliminazione $O(\log n)$

2.0.6 Hash table

Implementazione basata su array, dove l'elemento con chiave k è memorizzato nel $k -esimo$ "slot" dell'array.

- Ricerca caso medio $O(1)$ Ricerca caso pessimo $O(n)$
- Inserimento caso medio $O(1)$ Inserimento caso pessimo $O(n)$
- Eliminazione caso medio $O(1)$ Eliminazione caso pessimo $O(n)$

2.1 Riepilogo

	search		insert		delete	
	Medio	Pessimo	Medio	Pessimo	Medio	Pessimo
Array ordinato	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Lista non ordinata	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$
ABR	$O(\log n)$	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(n)$
Albero AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Albero 2-3	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Tabella Hash	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$

3 Algoritmi di ordinamento

3.0.1 ordinamento in loco:

L'algoritmo permuta gli elementi direttamente nell'array originale, senza usare un altro array di appoggio.

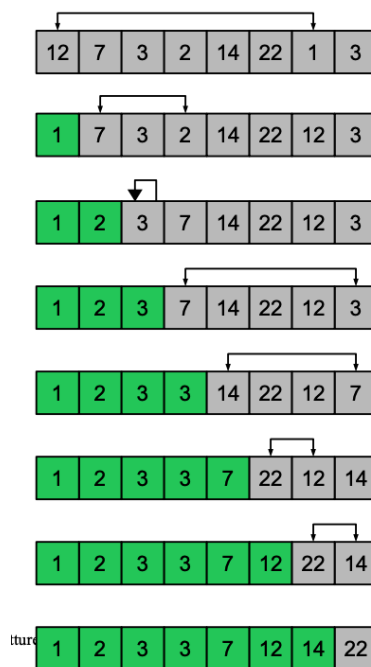
3.0.2 ordinamento stabile:

L'algoritmo preserva l'ordine con cui elementi con la stessa chiave compaiono nell'array originale.

3.1 Selection sort

Cerca il minimo in $A[k + 1..n]$ e spostalo in posizione $k + 1$.
La complessità di questo algoritmo è pari a:

$$O(n^2)$$

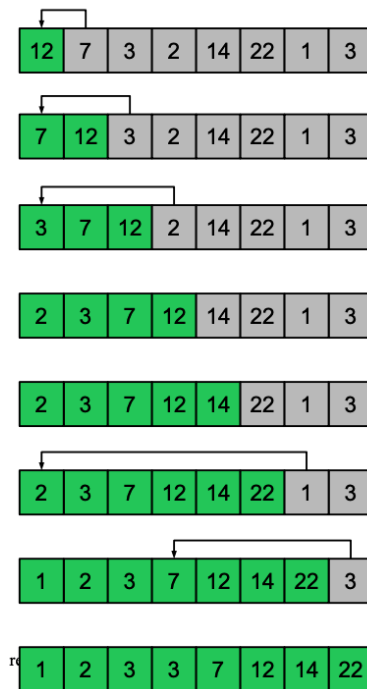


3.2 Insertion sort

Inserisco l'elemento di posizione $k+1$ nella **posizione corretta** all'interno dei primi k elementi ordinati. Al termine del passo k , il vettore ha le prime k componenti ordinate.

Il costo computazionale di questo algoritmo è:

$$O(n^2)$$

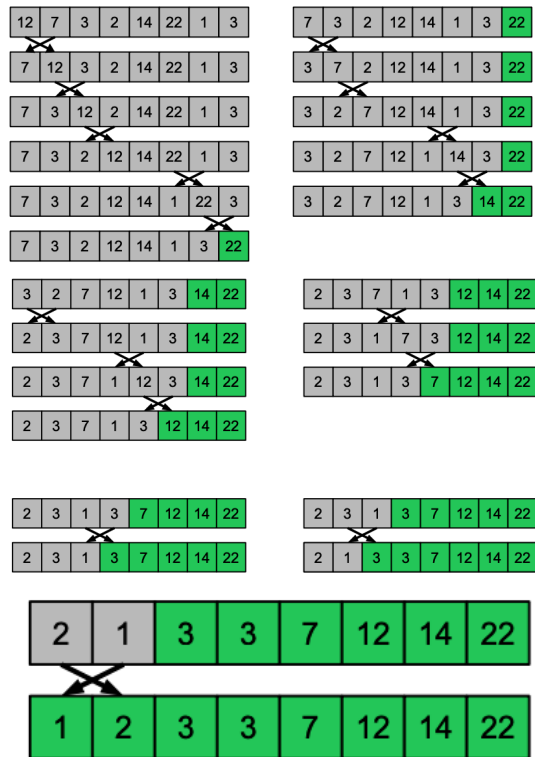


3.3 Bubble sort

Ad ogni scansione scambia le coppie di elementi adiacenti che non sono nell'ordine corretto.

Ad ogni scansione **scambia le coppie di elementi adiacenti**. Dopo la prima scansione, l'elemento massimo occupa l'ultima posizione, dopo la k-esima scansione, i k elementi massimi occupano la posizione corretta in fondo all'array. Nel caso *pessimo* – *ottimo* bubble Sort ha costo:

$$\Theta(n^2) \quad \Theta(n)$$



3.4 Quick sort

Scegli un elemento x del vettore v , e **partiziona il vettore in due parti** considerando gli elementi $\leq x$ e quelli $> x$

Ordina ricorsivamente le due parti.

Restituisci il risultato concatenando le due parti ordinate.

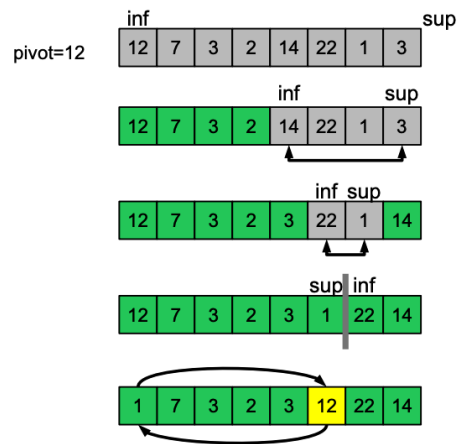
3.4.1 Partizionamento

Manteniamo due indici, inf e sup , che vengono fatti **scorrere dalle estremità** del vettore verso il centro. Quando entrambi (inf e sup) non possono essere fatti avanzare verso il centro, si **scambia** $A[inf]$ e $A[sup]$.

Il costo quick sort: Dipende dal partizionamento:

caso peggiore: $\Theta(n^2)$

caso migliore: $\Theta(n \log n)$



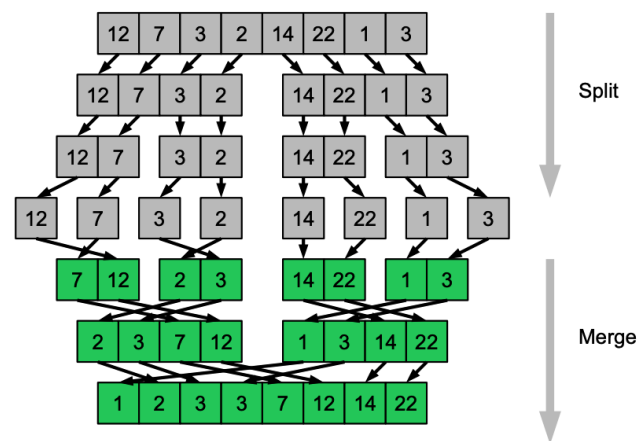
3.5 Merge Sort

Questo algoritmo **divide** $A[]$ in **due meta'** $A1[]$ e $A2[]$ (senza permutare) di dimensioni uguali;

Applica **ricorsivamente** Merge Sort a $A1[]$ e $A2[]$.

Fonde (merge) gli array ordinati $A1[]$ e $A2[]$ per ottenere l'array $A[]$ ordinato.

Merge Sort: esempio



3.6 Heapsort

Funzionamento:

1. Costruire un **max-heap** a partire dal vettore $A[]$ originale, mediante l'operazione **heapify()**
2. Estrarre il **massimo** ($findMax() + deleteMax()$)
3. **Inserire** il massimo in ultima posizione di $A[]$.
4. **Ripetere** il punto 2. finché lo heap diventa vuoto

3.6.1 Albero binario perfetto

Un albero binario è **perfetto** se:

- Tutte le foglie hanno la stessa altezza h
- Nodi interni hanno grado 2

Un albero perfetto ha altezza $h \simeq \log N$

Il numero di nodi è $N = \text{nod}i = 2^{h+1} - 1$

3.6.2 Albero binario completo

Un albero binario è **completo** se:

- Tutte le foglie hanno profondità h o $h-1$
- Tutti i nodi a livello h sono “accatastati” a sinistra
- Tutti i nodi interni hanno grado 2, eccetto al più uno

3.6.3 Max-heap

Un albero binario completo è un albero **max-heap** sse:

- Ad ogni nodo i viene associato un valore $A[i]$
- $A[\text{Parent}(i)] \geq A[i]$

3.6.4 Min-heap

Un albero binario completo è un albero **min-heap** sse:

- Ad ogni nodo i viene associato un valore $A[i]$
- $A[\text{Parent}(i)] \leq A[i]$

3.7 Operazioni su array heap

3.7.1 findMax()

Individua il valore massimo contenuto in uno heap.

Il massimo è sempre la radice, ossia $A[1]$.

L'operazione ha costo $\Theta(1)$.

3.7.2 fixHeap()

Ripristinare la proprietà di max-heap.

Supponiamo di rimpiazzare la radice $A[1]$ di un max-heap con un valore qualsiasi, vogliamo fare in modo che $A[]$ diventi nuovamente uno heap.

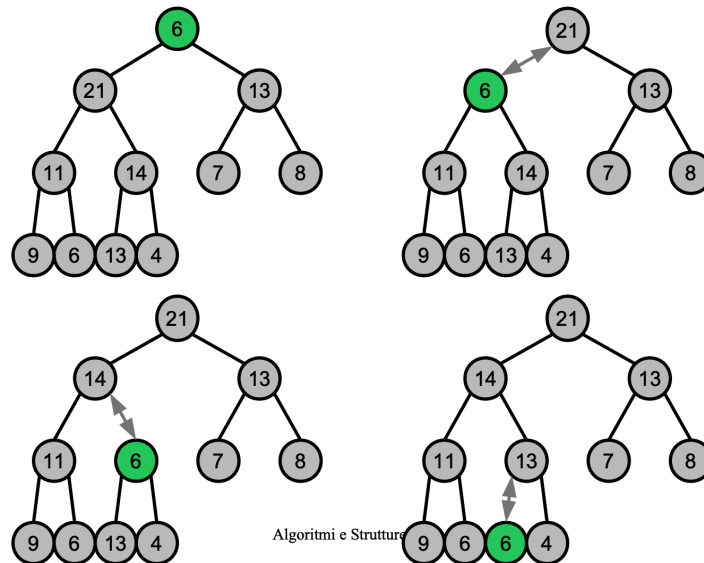
3.7.3 heapify()

Costruire uno heap a partire da un array privo di alcun ordine.

3.7.4 deleteMax()

Rimuovi l'elemento massimo da un maxheap $A[]$.

3.7.5 Esempio heapSort:



Algoritmi di ordinamento: sommario

- Abbiamo visto diversi algoritmi di ordinamento:

- **Selection Sort**: ottimo/medio/pessimo $\Theta(n^2)$
- **Insertion Sort**: ottimo/medio/pessimo $\Theta(n^2)$
- **Bubble Sort**: ottimo $\Theta(n)$, (medio)/pessimo $\Theta(n^2)$
- **Quick Sort**: ottimo $\Theta(n \log n)$, medio $\Theta(n \log n)$, pessimo $\Theta(n^2)$
- **Merge Sort**: ottimo/medio/pessimo $\Theta(n \log n)$ (non in-loco)
- **Heap Sort**: ottimo/medio/pessimo $\Theta(n \log n)$

Esercizio: come modificare per avere caso ottimo $\Theta(n)$?

- Nota:

- Tutti questi algoritmi sono basati su confronti
 - le decisioni sull'ordinamento vengono prese in base al confronto ($<$, $=$, $>$) fra due valori

Esercizio: perché il caso medio è $\Theta(n^2)$?

4 Tecniche lineari di ordinamento

4.1 Counting Sort

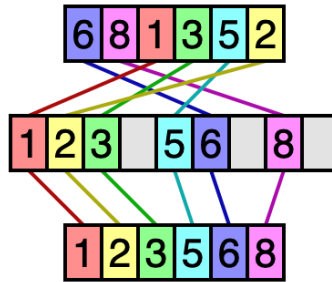
I valori di $A[0..n-1]$ appartengono all'intervallo $[0, k-1]$ (ciascun valore può comparire zero o più volte).

Costruisco un array $Y[0, k-1]$; $Y[i]$ conta il numero di volte in cui il valore i compare in A .

Ricolloco i valori così ottenuti in A .

Counting Sort: Costo

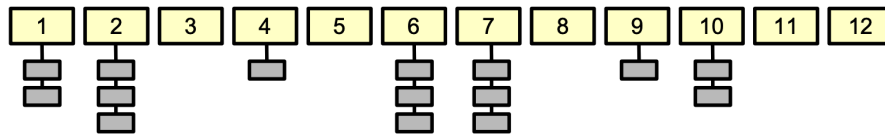
$$O(\max\{n, k\}) = O(n + k) = O(n)$$



4.2 Bucket Sort

Cosa succede se i valori da ordinare non sono numeri interi, ma record associati ad una chiave?

Possiamo usare liste concatenate.



Bucket Sort: Costo

$$O(n + k)$$

4.3 Radix Sort

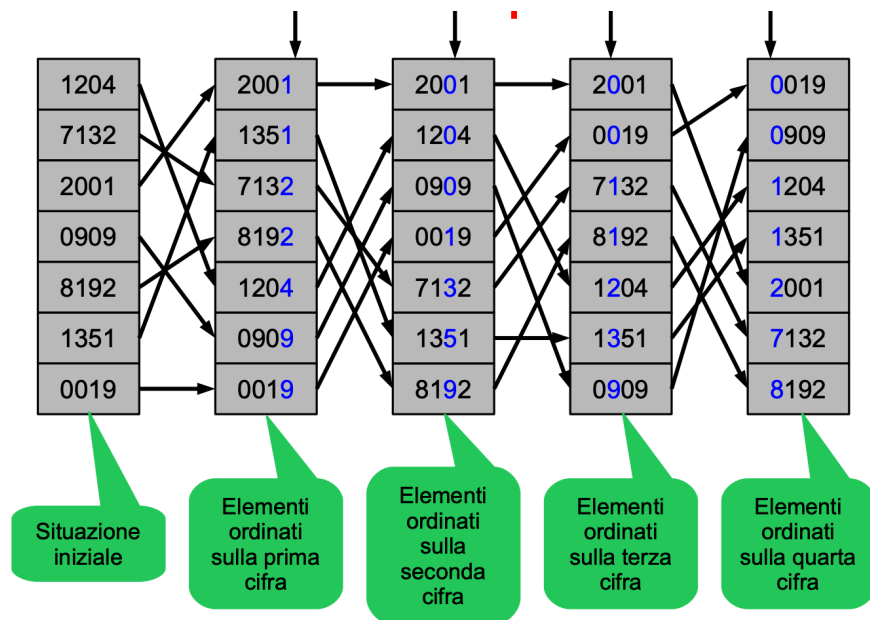
Supponiamo di voler ordinare n numeri con 4 cifre decimali.

Questo richiederebbe $n + 10000$ operazioni; se $n \log n < n + 10000$, questo non sarebbe conveniente.

Prima ordino in base alla cifra delle **unità**.

Poi ordino in base alla cifra delle **decine**.

Poi ordino in base alla cifra delle **centinaia**.



5 Selezione del k-esimo

Consideriamo il seguente problema:

Selezione del k-esimo minimo: dato un array $A[1..n]$ di valori distinti e un valore $1 \leq k \leq n$, trovare l'elemento che è maggiore di esattamente $k - 1$ elementi.

Mediano: il valore che occuperebbe la posizione $(n/2)$ se l'array fosse ordinato.

I **motori di ricerca** producono molti risultati a fronte di una singola query. I risultati vengono mostrati in pagine, in ordine decrescente di rilevanza. È **inutile ordinare tutti i risultati** in base alla rilevanza.

Verifichiamo ora i costi computazionali dei singoli casi:

Ricerca del minimo:

$$T(n) = n - 1 = \Theta(n)$$

Ricerca del secondo minimo:

$$T(n) = 2n - 3 = \Theta(n)$$

Selezione del k-esimo elemento:

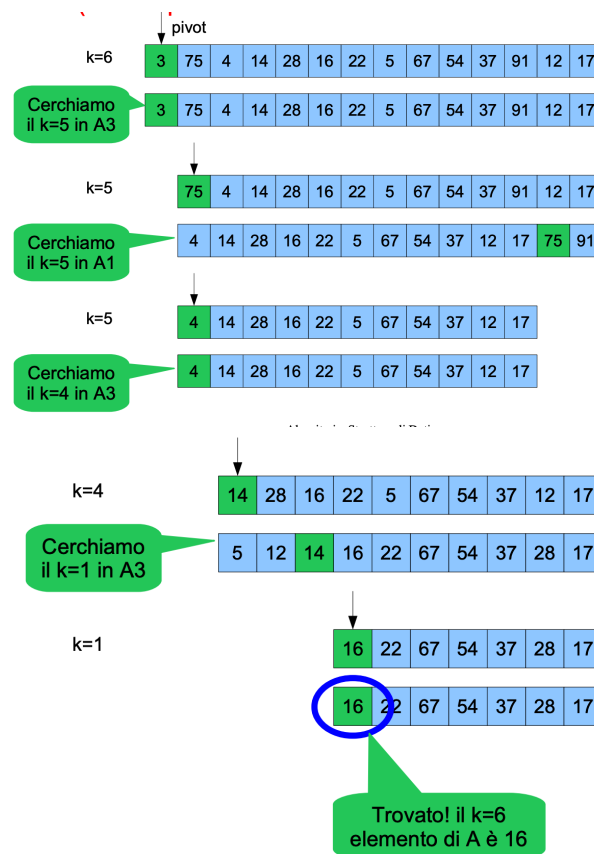
$$T(n) = \Theta(kn)$$

Selezione del valore mediano:

$$T(n) = O(n + k \log n) = O(n + (n/2) \log n) = O(n \log n)$$

5.0.1 Adattamento di quicksort al problema della selezione:

In questo modo divido il mio array in più partizioni, andando a eliminare quelle inutili.



5.0.2 Analisi dell'algoritmo quickSelect()

Costo nel caso ottimo:

$$T(n) = T(n/2) + n = \Theta(n)$$

Costo nel caso pessimo:

$$T(n) = T(n-1) + n = \Theta(n^2)$$

Costo nel caso medio:

$$T(n) \leq 4n$$

6 Code con priorità

Le code con priorità sono strutture dati che mantengono il minimo (massimo) in un insieme dinamico di chiavi.

$$coda = key|elem$$

Sono presenti due possibili implementazioni:

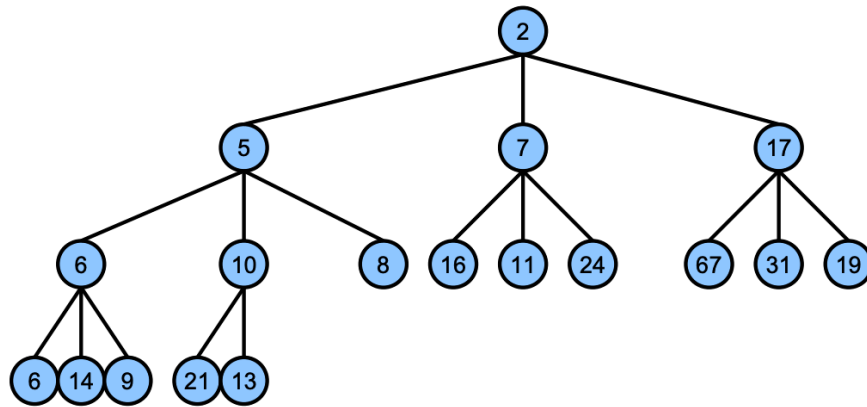
- d-heap
- heap binomiali e heap di fibonacci

6.0.1 d-heap

Un d-heap è un albero d-ario con le seguenti proprietà:

1. Un d-heap di altezza h è perfetto almeno fino alla profondità $h-1$; le foglie al livello h sono accatastate a sinistra.
2. Ciascun nodo v contiene una *chiave*(v) e un elemento *elem*(v). Le chiavi appartengono ad un dominio totalmente ordinato.
3. Ogni nodo diverso dalla radice ha chiave non inferiore (\geq) a quella del padre.

Esempio d-heap: $d = 3$



Un d-heap con n nodi ha **altezza** $O(\log_d n)$

Riepilogo costi per d-heap

- `findMin()` \rightarrow elem $O(1)$
- `insert(elem e, chiave k)` $O(\log_d n)$
- `delete(elem e)` $O(d \log_d n)$
- `deleteMin()` $O(d \log_d n)$
- `increaseKey(elem e, chiave d)` $O(d \log_d n)$
- `decreaseKey(elem e, chiave d)` $O(\log_d n)$

7 Union find

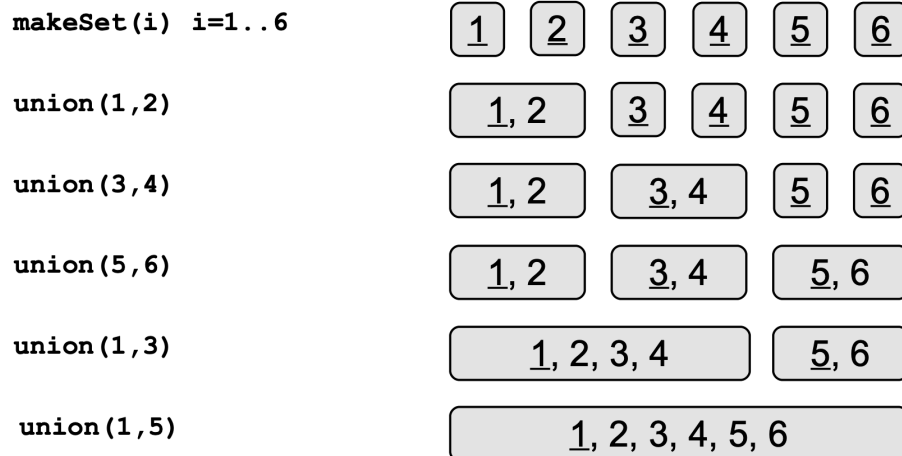
L'**union find** è una struttura dati basata sugli insiemi.

Essa può creare un insieme a partire da un singolo elemento, unire due insiemi, identificare l'insieme a cui appartiene un elemento. Gli insiemi contengono complessivamente $n \leq k$ elementi. Ogni insieme è identificato da un rappresentante univoco.

Operazioni Union find:

- *makeSet(elem x)*: Crea un insieme il cui unico elemento (e rappresentante) è x . Esso non deve appartenere ad un altro insieme esistente.
- *find(elem x) → name*: Restituisce il rappresentante dell'unico insieme contenente x .
- *union(name x, name y)*: Unisce i due insiemi rappresentati da x e da y . Assumiamo che il nome del nuovo insieme sia x . I vecchi insiemi devono essere distrutti.

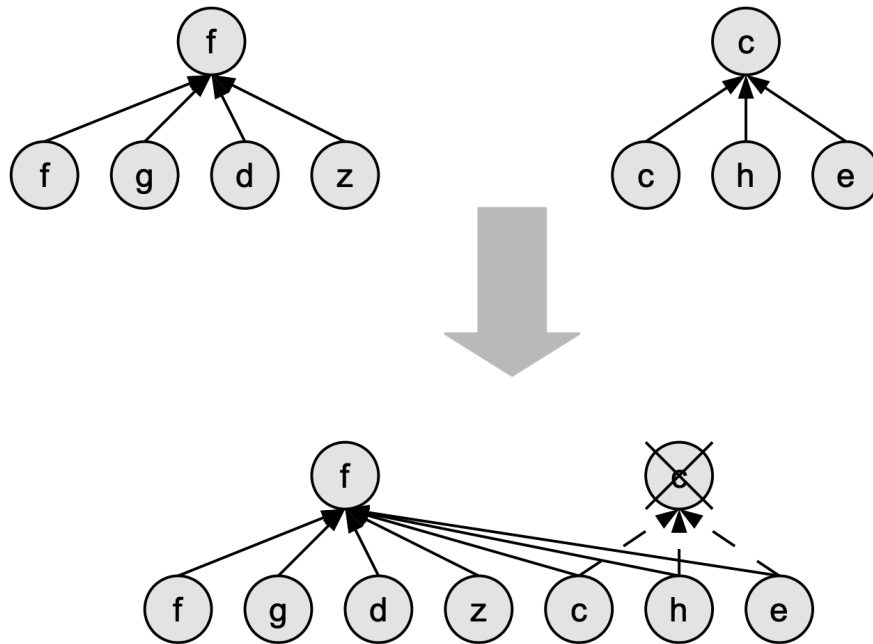
Esempio:



7.1 QuickFind

Ogni insieme viene rappresentato con un **albero** di altezza uno.

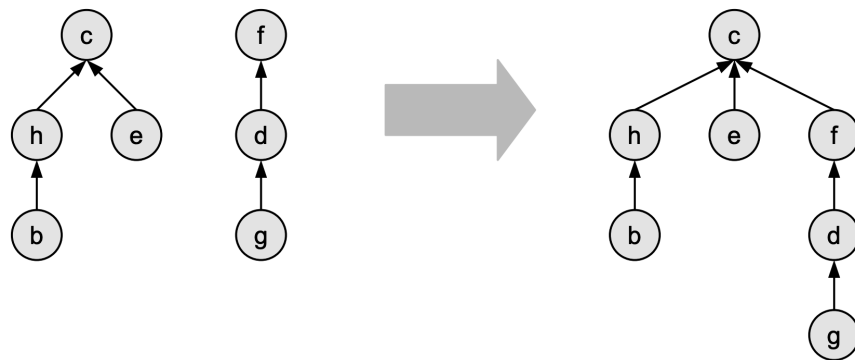
- Le foglie dell'albero contengono gli elementi dell'insieme.
- Il rappresentante è la radice.



7.2 QuickUnion

Implementazione basata su foresta.

- Si rappresenta ogni insieme tramite un albero radicato generico.
- Ogni nodo dell'albero contiene l'oggetto e il puntatore al padre.
- Il rappresentante è la radice.



Riepilogo

	QuickFind	QuickUnion
makeSet	$O(1)$	$O(1)$
union	$O(n)$	$O(1)$
find	$O(1)$	$O(n)$

E' conveniente utilizzare **QuickFind** quando le *union()* sono rare e le *find()* frequenti.

E' conveniente utilizzare **QuickUnion** quando le *find()* sono rare e le *union()* frequenti.

8 Divide et Impera

- Divide-et-impera
 - Un problema viene suddiviso in sotto-problemi, che vengono risolti ricorsivamente (top-down).
- Algoritmi greedy
 - Ad ogni passo si fa sempre la scelta che in quel momento appare ottima; le scelte fatte non vengono mai disfatte
- Programmazione dinamica
 - La soluzione viene costruita (bottom-up) a partire da un insieme di sotto-problemi

Tecnica divisa in **3 fasi** fondamentali:

1. **Divide:** Dividi il problema in sotto-problemi indipendenti, di dimensioni *minori*.
2. **Impera:** Risolvi i sotto-problemi ricorsivamente.
3. **Combina:** Unisci le soluzioni dei sottoproblemi per costruire la soluzione del problema di partenza

8.1 Algoritmi greedy

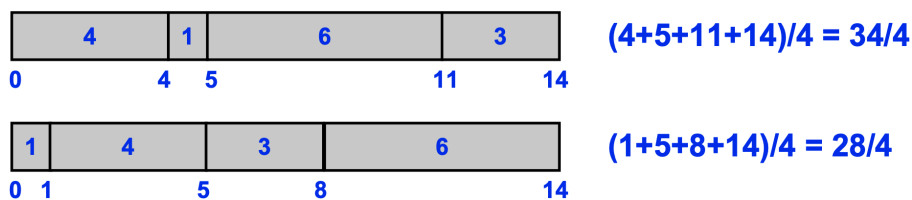
Quando applicare la tecnica greedy?

- Quando è possibile dimostrare che esiste una **scelta ingorda**.
 - Fra le molte scelte possibili, se ne può facilmente individuare una che porta sicuramente alla soluzione ottima.
- Quando il problema ha **sottostruttura ottima**.
 - “Fatta tale scelta, resta un sottoproblema con la stessa struttura del problema principale”.

8.2 Algoritmo di scheduling

Algoritmo che si basa in base al tempo medio di esecuzione:

- 1 processore, n job p_1, p_2, \dots, p_n .
- Ogni job p_i ha un tempo di esecuzione $t[i]$.
- Minimizzare il tempo medio di completamento.



8.3 Codifica di Huffman

Questa codifica viene utilizzata per risolvere il problema di compressione, (compressione di un file). Viene utilizzata una tecnica detta **codifica di caratteri**

- Si usa una **funzione di codifica f**: $f(c) = x$
 1. c è un carattere preso da un alfabeto Σ
 2. x è una rappresentazione binaria del carattere c
 3. c è rappresentato da x in modo efficiente
- Una sequenza di caratteri $c_1 c_2 c_n$ viene codificata con la sequenza di bit $f(c_1)f(c_2)f(c_n)$
- data una qualsiasi codifica, deve essere sempre possibile decodificarla durante la lettura sequenziale bit-dopo-bit.

Dobbiamo utilizzare una codifica che minimizza la dimensione del nostro file.

8.3.1 Codifica a lunghezza fissa

- Possibili car.: 'a' 'b' 'c' 'd' 'e' 'f'
- frequenze: **45%** **13%** **12%** **16%** **9%** **5%**

- Supponiamo di avere un file di n caratteri.
- Codifica tramite ASCII (8 bit per carattere).
- Codifica basata sull'alfabeto (3 bit per carattere).

8.3.2 Codifica a lunghezza variabile

- Caratteri: 'a' 'b' 'c' 'd' 'e' 'f'
- Codifica: **0** **101** **100** **111** **1101** **1100**
- Costo totale:
 $(0.45*1+0.13*3+0.12*3+0.16*3+0.09*4+0.05*4)*n=2.24n$

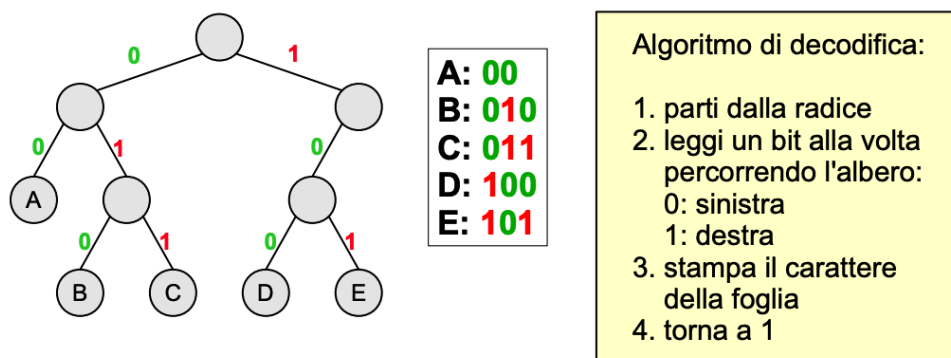
- Codifica a lunghezza variabile
- **codice a prefisso** (senza prefissi):
 - **nessun codice è un prefisso di un altro codice.**
 - Condizione richiesta per permettere sempre la decodifica durante la lettura bit-dopo-bit.

In questo modo avendo dei **prefissi univoci**, non appena troviamo una sequenza di bit possiamo risalire alla decodifica della parola. In questo modo **lettere frequenti** saranno composte da **pochi bit**.

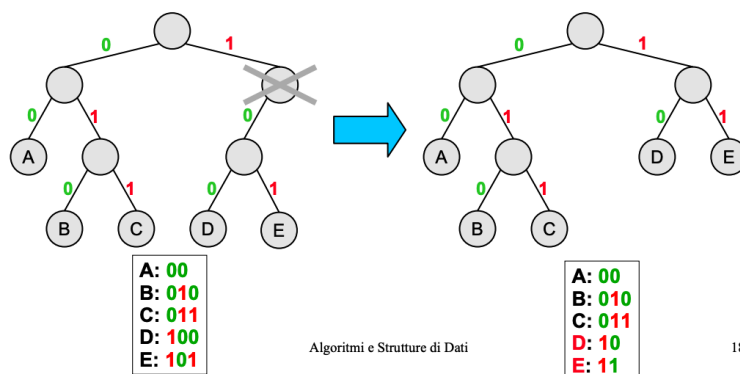
8.3.3 Codici di Huffman

Rappresentazione del codice come un albero binario

- Figlio sinistro: 0 Figlio destro: 1
- Caratteri dell'alfabeto sulle foglie



Questo esempio per essere ottimizzato deve avere anche un figlio destro al primo livello di profondità.



Il principio del codice di Huffman è:

- **Minimizzare la lunghezza dei caratteri** che compaiono più frequentemente.
- Assegnare ai caratteri con la frequenza i codici corrispondenti ai percorsi più lunghi all'interno dell'albero.

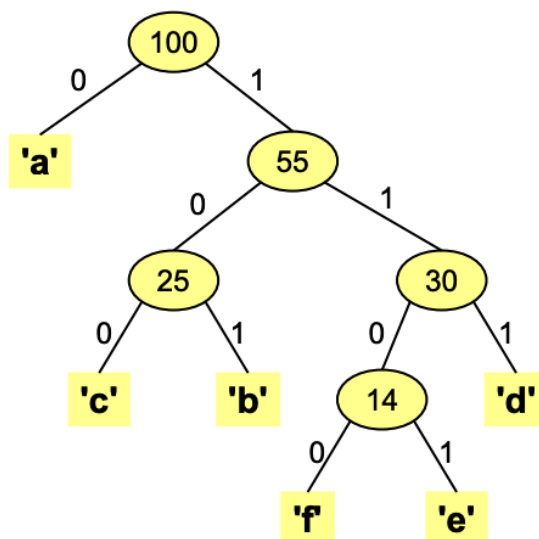
1. Inizialmente costruiamo una lista ordinata di nodi, in cui ogni nodo contiene un carattere e il numero di volte in cui quel carattere compare nel file.

"f" : 5 "e" : 9 "c" : 12 "b" : 13 "d" : 16 "a" : 45

2. Rimuovere i due nodi con frequenze minori.
3. Colregarli ad un nodo padre etichettato con la frequenza combinata (sommata).

"c" : 12 "b" : 13 "d" : 16 "a" : 45

4. Aggiungere il nodo combinato alla lista, mantenendola ordinata in base alla frequenza.



Vantaggi:

- Semplici da programmare
- Solitamente efficienti
- Quando è possibile dimostrare la proprietà di scelta greedy danno la soluzione ottima

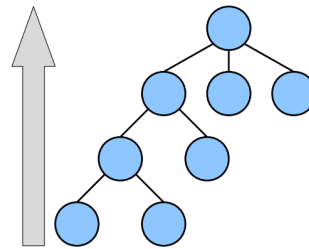
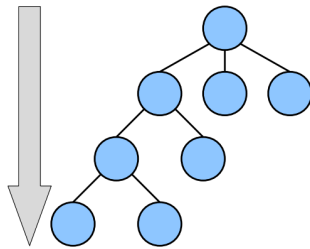
- La soluzione sub-ottima può essere accettabile

Svantaggi:

- Non tutti i problemi ammettono una soluzione greedy
- Quindi, in certi casi gli algoritmi greedy non possono essere usati se si vuole la soluzione ottima

9 Programmazione dinamica

- **Divide-et-impera**
 - Tecnica ricorsiva
 - Approccio **top-down**
 - Vantaggiosa quando i sottoproblemi sono **indipendenti**
- **Programmazione dinamica**
 - Tecnica iterativa
 - Approccio **bottom-up**
 - Vantaggiosa quando ci sono sottoproblemi **ripetuti**



Algoritmi e Strutture di Dati

4

1. **Sottostruttura ottimale**, deve essere possibile combinare le soluzioni dei sottoproblemi.
2. **Sottoproblemi ripetuti**, che ricompaiono costantemente.

9.1 Distanza di Levenshtein

Tecnica utilizzata dai correttori ortografici. Basata su:

- Concetto di edit distance:
 - Numero di operazioni di “editing” che sono necessarie per trasformare una stringa S in una nuova stringa T .
- Trasformazioni ammesse:
 - Lasciare immutato il carattere corrente (costo 0).
 - Cancellare un carattere (costo 1).
 - Inserire un carattere (costo 1).
 - Sostituire il carattere corrente con uno diverso (costo 1).
- Dopo ciascuna operazione ci si sposta sul carattere successivo:
 - Si inizia dal primo carattere di S .

La **distanza di levenshtein** tra $S[1..n]$ e $T[1..m]$ è il **costo minimo** tra tutte le sequenze di operazioni di editing che trasformano S in T .

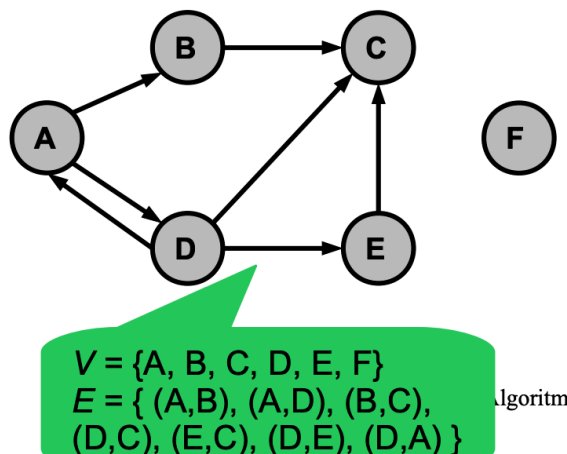
Esempio:

- Determinare il numero minimo di operazioni di editing necessarie per trasformare il prefisso $S[1..i]$ di S nel prefisso $T[1..j]$ di T .
- La definizione della soluzione è data da $L[1..j]$. - La distanza di Levenshtein tra $S[1..n]$ e $T[1..m]$ è il valore $L[n, m]$.

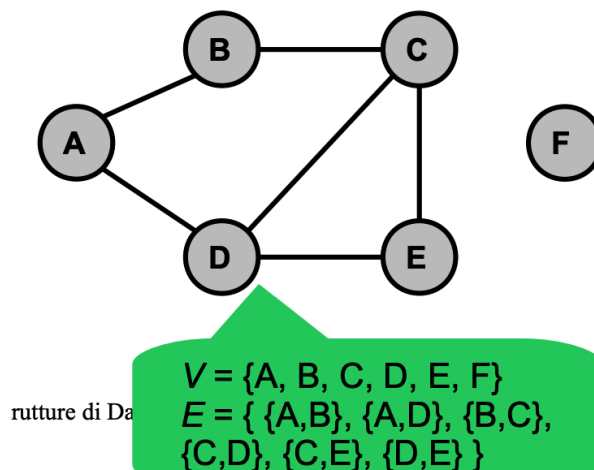
10 Grafi

10.1 Grafi orientati e non orientati

- Un **Grafo orientato** G è una coppia (V, E) dove:
 - Insieme finito dei **vertici** V
 - Insieme degli **archi** E : relazione binaria tra vertici



- Un **grafo non orientato** G è una coppia (V, E) dove:
 - Insieme finito dei **vertici** V
 - Insieme degli **archi** E : coppie non ordinate

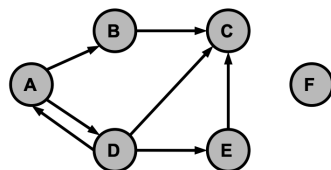


10.2 Problemi sui grafi

- Visite
 - Visite in ampiezza
 - Visite in profondità
- Alberi di copertura minimi
- Cammini minimi
 - Da singola sorgente
 - Fra tutte le coppie dei vertici

10.2.1 Incidenza e adiacenza

- In un grafo orientato l'arco (v, w) è **incidente** da v a w
- Un vertice w è **adiacente** a v se e solo se $(v, w) \in E$
- In un grafo non orientato la relazione di adiacenza tra vertici è simmetrica



(A, B) è incidente da A a B
 (A, D) è incidente da A a D
 (D, A) è incidente da D a A

B è adiacente ad A
C è adiacente a B, D, E
A è adiacente a D e viceversa
B non è adiacente a D, C
F non è adiacente ad alcun vertice

- NumVertici() → intero
- NumArchi() → intero
- grado(vertex v) → intero
- archiIncidenti(vertex v) → (arco, arco, ... arco)
- estremi(arco e) → (vertex, vertex)
- opposto(vertex x, arco e) → vertex
- sonoAdiacenti(vertex x, vertex y) → booleano
- aggiungiVertice(vertex v)
- aggiungiArco(vertex x, vertex y)
- rimuoviVertice(vertex v)
- rimuoviArco(arco e)

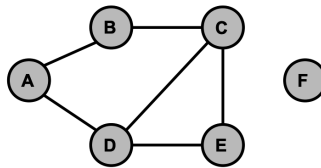
n = vertici

m = numero archi

10.3 Rappresentazioni di grafi

- Liste di archi:

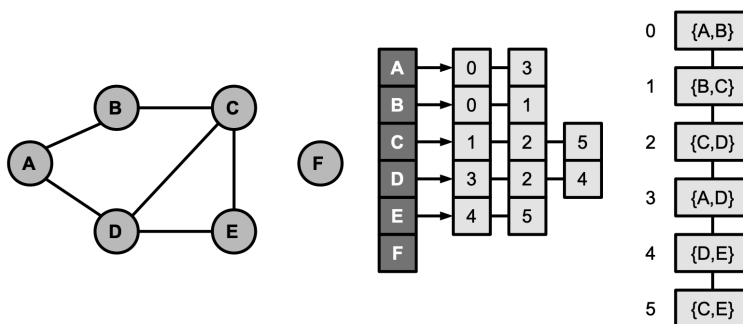
- grado = $O(m)$
- archiIncidenti = $O(m)$
- sonoAdiacenti = $O(m)$
- aggiungiVertice = $O(1)$
- aggiungiArco = $O(1)$
- rimuoviVertice = $O(m)$
- rimuoviArco = $O(1)$



δ = grado

- Liste di incidenza

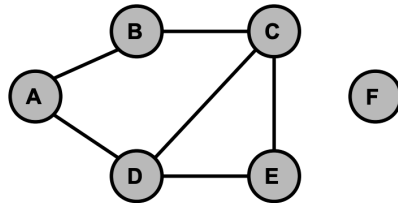
- grado = $O(\delta(n))$
- archiIncidenti = $O(\delta(n))$
- sonoAdiacenti = $O(\min \delta(x), \delta(y))$
- aggiungiVertice = $O(1)$
- aggiungiArco = $O(1)$
- rimuoviVertice = $O(m)$
- rimuoviArco = $O(\delta(x) + \delta(y))$



- Matrice di adiacenza

- grado = $O(n)$
- archiIncidenti = $O(n)$
- sonoAdiacenti = $O(1)$
- aggiungiVertice = $O(n^2)$
- aggiungiArco = $O(1)$
- rimuoviVertice = $O(n^2)$
- rimuoviArco = $O(1)$

$$M(u, v) = \begin{cases} 1 & \text{se } \{u, v\} \in E \\ 0 & \text{altrimenti} \end{cases}$$



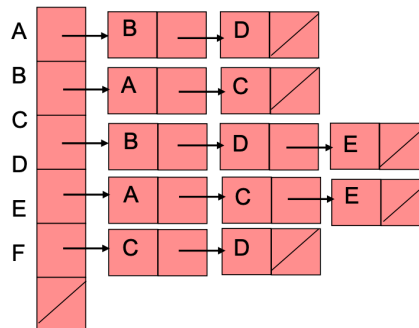
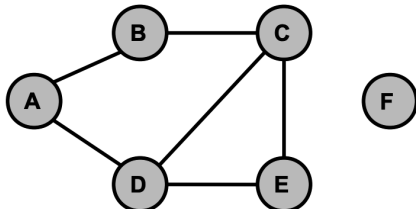
$$M = \begin{matrix} & \begin{matrix} A & B & C & D & E & F \end{matrix} \\ \begin{matrix} A \\ B \\ C \\ D \\ E \\ F \end{matrix} & \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{matrix}$$

Spazio: $\Theta(|V|^2)$

- Liste di adiacenza

- grado = $O(\delta(v))$
- archiIncidenti = $O(\delta(v))$
- sonoAdiacenti = $O(\min \delta(x), \delta(y))$
- aggiungiVertice = $O(1)$
- aggiungiArco = $O(1)$
- rimuoviVertice = $O(m)$
- rimuoviArco = $O(\delta(x) + \delta(y))$

$$v.\text{adj} = \{ w \mid \{v, w\} \in E \}$$



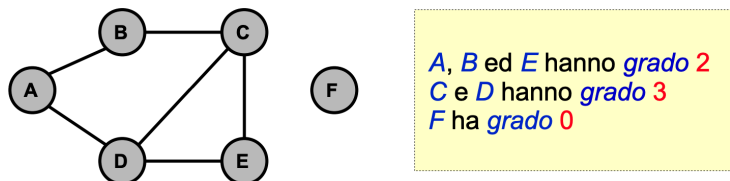
Spazio: $\Theta(|V| + |E|)$

10.4 Grafi pesati

In alcuni casi ogni arco ha un **peso** (o **costo**) associato. Il costo può essere determinato tramite una funzione di costo $c: E \in R$, dove R è l'insieme dei numeri reali. Quando tra due vertici non esiste un arco, si dice che il costo è **infinito**.

10.5 Grado

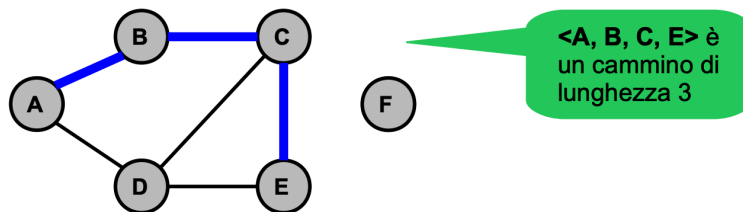
In un **grafo non orientato**, il **grado** di un vertice è il **numero di archi** che partono da esso.



- In un **grafo orientato**, il grado entrante (uscente) di un vertice è il **numero di archi incidenti** in (da) esso
- In un **grafo orientato** il grado di un vertice è la **somma** del suo grado entrante e del suo grado uscente

10.6 Cammini

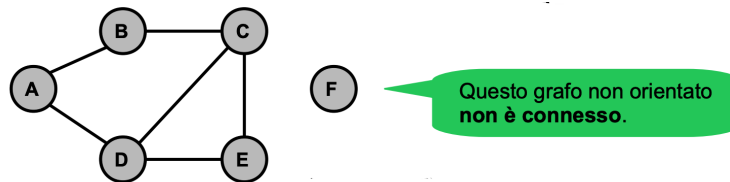
La lunghezza del cammino è il numero di archi attraversati.



Un cammino si dice **semplice** se tutti i suoi vertici sono **distinti** (compaiono una sola volta nella sequenza).

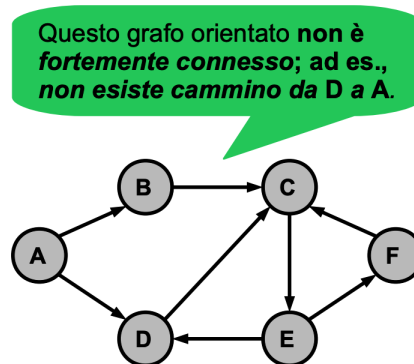
10.7 Grafi connessi

Se G è un grafo **non orientato**, diciamo che G è **connesso** se esiste un cammino da ogni vertice ad ogni altro vertice.



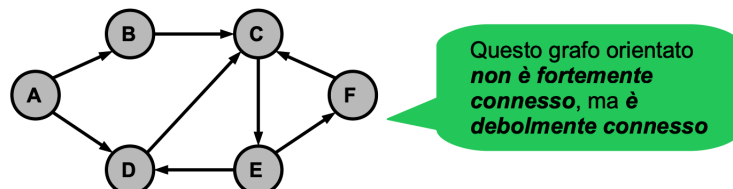
10.7.1 Grafo fortemente connesso

Se G è un grafo **orientato**, diciamo che G è **fortemente connesso** se esiste un cammino da ogni vertice ad ogni altro vertice.



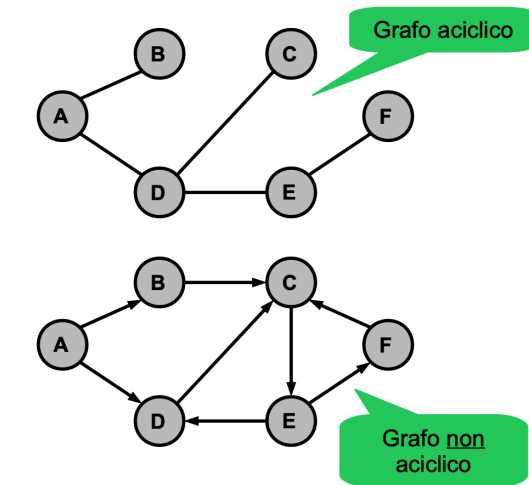
10.7.2 Grafo debolmente connesso

Se G è un grafo **orientato** che non è fortemente connesso, ma la sua versione non orientata è connessa, diciamo che G è **debolmente connesso**.



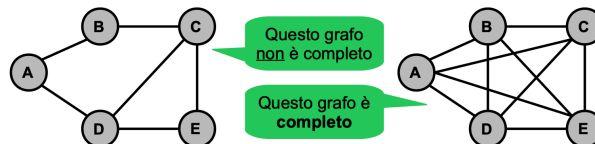
10.8 Grafi aciclici

Un grafo senza cicli semplici è detto **aciclico**. Un grafo orientato aciclico è chiamato **DAG** (Directed Acyclic Graph).



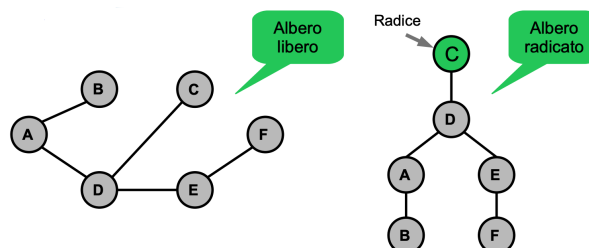
10.9 Grafo completo

Un **grafo non orientato completo** è un grafo non orientato che ha un arco tra ogni coppia di vertici.



10.10 Alberi

Un **albero libero** è un grafo non orientato connesso, aciclico. Se un vertice è detto radice, otteniamo un **albero radicato**.



11 Algoritmi di Visita di grafi

- **Visita in ampiezza** (breadth-first search)
 - Visita i nodi “espandendo” la frontiera fra nodi scoperti / da scoprire
 - Es: Cammini di lunghezza minima da singola sorgente
- **Visita in profondità** (depth-first search)
 - Visita i nodi andando il “più lontano possibile” nel grafo
 - Es: Componenti fortemente connesse, ordinamento topologico

11.0.1 Vertici del grafo

Ogni vertice del grafo può essere:

- **inesplorato**: Il vertice non è ancora stato incontrato.
- **aperto**: l'algoritmo ha incontrato il vertice la prima volta.
- **chiuso**: il vertice è stato visitato completamente (tutti gli archi incidenti sono stati esplorati).

11.1 Algoritmo di visita generico

```
algoritmo visita(G, s)→albero
  rendi "non marcati" tutti i vertici
  T := s
  F := { s }
  "marca" il vertice s
  while (F ≠ ∅) do
    u := F.extract()
    "visita il vertice u"
    for each v adiacente a u do
      if (v non è marcato) then
        marca il vertice v
        T := T ∪ v
        F.insert(v)
        v.parent := u
      endif
    endfor
  endwhile
  return T
```

- F è l'insieme **frontiera** (o **frangia**)
- Il funzionamento di *extract()* e *insert()* non è specificato
- T è l'albero che viene costruito dalla visita
- $v.parent$ è il padre di v nell'albero T

11.1.1 Complessità

- $O(n + m)$ liste di adiacenza
- $O(n^2)$ matrice di adiacenza

11.2 Algoritmo di visita in ampiezza

- Visitare i nodi a distanze crescenti dalla sorgente
- Generare un albero BF (breadth-first), cioè un albero contenente tutti i vertici
- Calcolare la distanza minima da s a tutti i vertici raggiungibili

```
algoritmo BFS(Grafo G, vertice s) → albero
for each v in V do v.mark := false
T := s
F := new Queue()
F.enqueue(s)
s.mark := true
s.dist := 0
while (F ≠ ∅) do
  u := F.dequeue()
  "visita il vertice u"
  for each v adiacente a u do
    if (not v.mark) then
      v.mark := true
      v.dist := u.dist + 1
      F.enqueue(v)
      v.parent := u
    endif
  endfor
endwhile
return T
```

- Insieme F gestito tramite una coda
- $v.mark$ è la marcatura del nodo v
- $v.dist$ è la distanza del nodo v dal vertice s

11

11.3 Algoritmo di visita in profondità

- Utilizzata per coprire l'intero grafo, non solo i nodi raggiungibili da una singola sorgente (diversamente da BFS)
- Informazioni aggiuntive sul tempo di visita

```
algoritmo DFS-visit(vertice u)
u.mark := gray;
time := time + 1;
u.dt := time;
for each v adiacente a u do
  if (v.mark = white) then
    v.parent := u;
    DFS-visit(v);
  endif
endfor
"visita il vertice u"
time := time + 1;
u.ft := time;
u.mark := black;
```

- Nodi bianchi = inesplorati
- Nodi grigi = aperti
- Nodi neri = chiusi

11.3.1 Proprietà della visita DFS

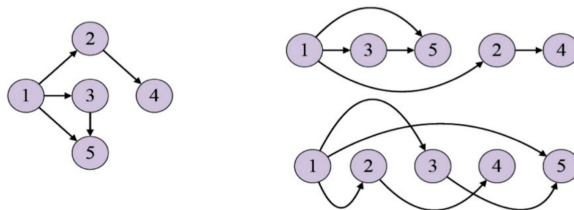
In una qualsiasi visita in profondità per ogni coppia di vertici u, v **una sola** delle seguenti condizioni è vera:

1. Gli intervalli $[u.dt, u.ft]$ e $[v.dt, v.ft]$ sono disgiunti (non sono discendenti)
2. L'intervallo $[u.dt, u.ft]$ è interamente contenuto in $[v.dt, v.ft]$ (u è discendente di v)
3. L'intervallo $[v.dt, v.ft]$ è interamente contenuto in $[u.dt, u.ft]$ (v è discendente di u)

11.4 Ordinamento topologico

Dato un DAG G (direct acyclic graph), un ordinamento topologico su G è un ordinamento lineare dei suoi vertici tale per cui:

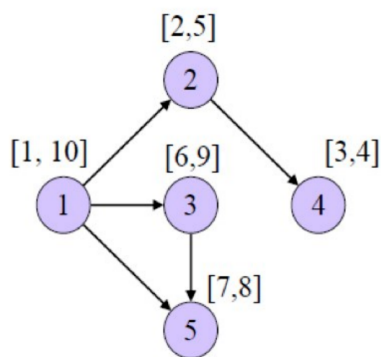
- Se G contiene l'arco (u, v) , allora u compare prima di v nell'ordinamento
- Per transitività, ne consegue che se v è raggiungibile da u , allora u compare prima di v nell'ordinamento



11.4.1 Algoritmo per ordinamento topologico

Algoritmo:

1. Si effettua una DFS
2. L'operazione di visita aggiunge il nodo alla testa di una lista "at finish time"
3. Restituire la lista di vertici



11.5 Collegare elementi minimizzando vincoli

Minimizzare la quantità di filo elettrico per collegare fra loro i diversi componenti.

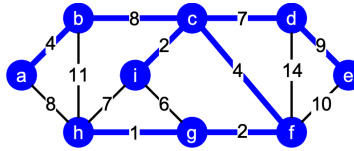
- albero di copertura (di peso) minimo.
- albero di connessione (di peso) minimo.
- minimum spanning tree.

11.5.1 Albero di copertura (spanning tree)

Dato un grafo $G = (V, E)$ non orientato e connesso, un albero di copertura di G è un sottografo $T = (V, E_T)$ tale che:

- T è un albero
- $E_T \subseteq E$

- T contiene tutti i nodi di G



Il **Minimum Spanning Tree** non è necessariamente unico.

11.6 Algoritmo generico

- Vediamo:
 - Un algoritmo **greedy** generico.
 - Due istanze di questo algoritmo: **Kruskal** e **Prim**
- L'idea è di **accrescere** un sottoinsieme T di archi in modo tale che venga rispettata la seguente condizione:
 - T è un sottoinsieme di qualche albero di copertura minimo
- Un arco u, v è detto **sicuro** per T se $T \cup \{u, v\}$ è ancora un sottoinsieme di qualche MST

```

Tree Generic-MST(Grafo  $G=(V,E,w)$ )
  Tree  $T \leftarrow$  Albero vuoto
  while  $T$  non forma un albero di copertura do
    trova un arco sicuro  $\{u, v\}$ 
     $T \leftarrow T \cup \{u, v\}$ 
  endwhile
  return  $T$ 

```

- **Archi blu**
 - sono gli archi che fanno parte del MST.
- **Archi rossi**
 - sono gli archi che non fanno parte del MST

Per caratterizzare gli archi sicuri dobbiamo introdurre alcune definizioni:

- Un **taglio** $(S, V - S)$ di un grafo non orientato $G = (V, E)$ è una partizione di V in due sottoinsiemi disgiunti

- Un arco u, v **attraversa il taglio** se $u \in S$ e $v \in V - S$
- Un taglio **rispetta** un insieme di archi T se nessun arco di T attraversa il taglio.
- Un arco che attraversa un taglio è **leggero** se il suo peso è minimo fra i pesi degli archi che attraversano un taglio.

1. Regola del taglio

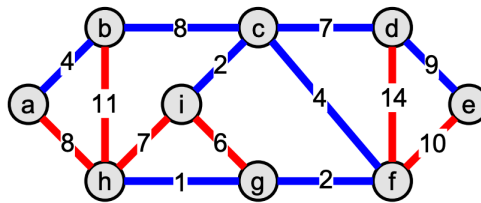
- Scegli un taglio in G che **non contenga archi blu**. Tra tutti gli archi non colorati che attraversano il taglio seleziona un arco di costo minimo e coloralo di blu

2. Regola del ciclo

- Scegli un ciclo semplice in G che **non contenga archi rossi**. Tra tutti gli archi non colorati del ciclo, seleziona un arco di costo massimo e coloralo di rosso

3. Metodo greedy

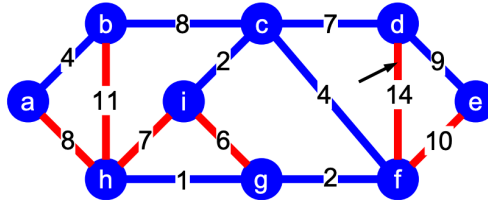
- Costruisce un MST applicando, ad ogni passo, una delle due regole precedenti (una qualunque, purché si possa usare)



11.6.1 Algoritmo di Kruskal

- Ingrandire sottoinsiemi disgiunti di un albero di copertura minimo connettendoli fra di loro fino ad avere l'albero finale
 - Inizialmente la **foresta di copertura** è composta da n alberi, uno per ciascun nodo, e nessun arco.
- Si considerano gli archi in ordine non decrescente di peso

- L'algoritmo è greedy perché ad ogni passo si aggiunge alla foresta un arco con il peso minimo



Costo computazionale:

n = vertici

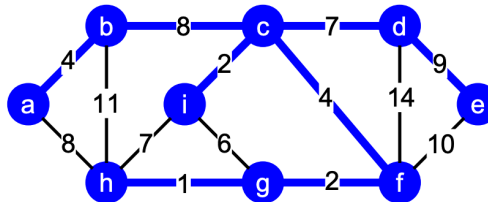
m = numero archi

$$O(m \log n)$$

11.7 Algoritmo di Prim

L'algoritmo di **Prim** procede mantenendo in un singolo albero T che viene fatto via via “crescere”.

- L'albero parte da un nodo arbitrario **r** (la **radice**) e cresce fino a quando ricopre tutti i vertici.
- Ad ogni passo viene aggiunto l'arco di peso minimo che collega un nodo già raggiunto dell'albero con uno non ancora raggiunto



Costo computazionale:

n = vertici

m = numero archi

$$O(m \log n)$$

12 Cammini minimi

Consideriamo un grafo orientato $G = (V, E)$ in cui ad ogni arco $(x, y) \in E$ sia associato un costo $w(x, y)$.

Data una coppia di nodi v_0 e v_k , vogliamo trovare (se esiste) un cammino $\pi_{v_0 v_k}^*$ di costo minimo tra tutti i cammini che vanno da v_0 a v_k .

Problemi simili da risolvere:

1. **Cammino di costo minimo fra una singola coppia di nodi u e v**

- Determinare, se esiste, un cammino di costo minimo π_{uv}^* da u verso v .

2. **Single-source shortest path**

- Determinare cammini di costo minimo da un nodo sorgente s a tutti i nodi raggiungibili da s .

3. **All-pairs shortest paths**

- Determinare cammini di costo minimo tra ogni coppia di nodi u, v

12.0.1 Proprietà (sottostruttura ottima)

Sia $G = (V, E)$ un grafo orientato con funzione costo w ; allora ogni sotto-cammino di un cammino di costo minimo in G è a sua volta un cammino di costo minimo.

12.1 Condizioni di Bellman Ford

- Per ogni arco (u, v) e per ogni vertice s , vale la seguente disuguaglianza

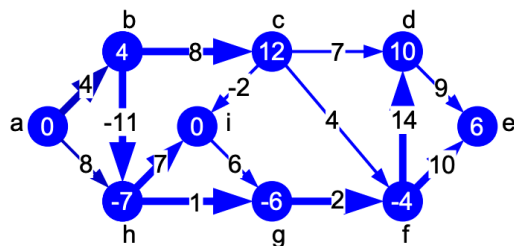
$$d_{sv} \leq d_{su} + w(u, v)$$

Dalla condizione di **Bellman** si può dedurre che l'arco (u, v) fa parte di un cammino di **costo minimo** π_{sv}^* se e solo se:

$$d_{sv} = d_{su} + w(u, v)$$

- Supponiamo di mantenere una stima $D_{sv} \leq d_{sv}$ della lunghezza del cammino di costo minimo tra s e v .
- Effettuiamo dei passi di “rilassamento”, riducendo progressivamente la stima finché si ha $D_{sv} = d_{sv}$.

if ($D_{su} + w(u, v) < D_{sv}$) **then** $D_{sv} \leftarrow D_{su} + w(u, v)$



Costo computazionale:

n = numero archi

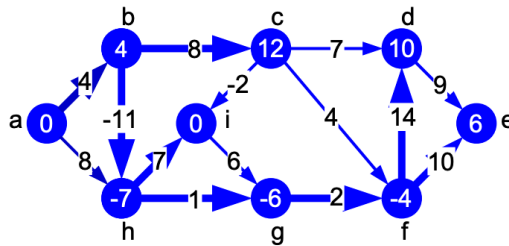
m = numero vertici

$$O(nm)$$

12.2 Dijkstra

Determina i cammini di **costo minimo** da singola sorgente nel caso in cui tutti gli archi abbiano costo ≥ 0 .

- Sia $G = (V, E)$ un grafo orientato con funzione costo w .
 - I costi degli archi devono essere ≥ 0 .
- Sia T una parte dell'albero dei cammini di costo minimo radicato in s
 - T rappresenta porzioni di cammini di costo minimo che partono da s .
- Allora l'arco (u, v) con $u \in V(T)$ e $v \notin V(T)$ che minimizza la quantità $d_{su} + w(u, v)$ appartiene ad un cammino minimo da s a v .



12.3 Algoritmo Dijkstra

```
double[1..n] Dijkstra(Grafo G=(V,E,w), int s)
int n ← G.numNodi();
int pred[1..n], v, u;
double D[1..n];
for v ← 1 to n do
  D[v] ← +∞;
  pred[v] ← -1;
endfor
D[s] ← 0;
CodaPriorita<int, double> Q; Q.insert(s, D[s]);
while (not Q.isEmpty()) do
  u ← Q.find(); Q.deleteMin();
  for each v adiacente a u do
    if (D[v] == +∞) then
      D[v] ← D[u] + w(u,v);
      Q.insert(v, D[v]);
      pred[v] ← u;
    elseif (D[u] + w(u,v) < D[v]) then
      Q.decreaseKey(v, D[v] - D[u] - w(u,v));
      D[v] ← D[u] + w(u,v);
      pred[v] ← u;
    endif
  endfor
endwhile
return D;
```

Trova e rimuovi il nodo con distanza minima

Somiglia all'algoritmo di Prim (MST), ma usa una priorit  diversa

Rendi $D[u] + w(u,v)$ la nuova distanza di v da s

12.3.1 Costo computazionale:

- L'inizializzazione ha costo $O(n)$.
- Le operazioni $find()$ e $deleteMin()$ hanno costo $O(\log n)$ e sono eseguite al più n volte.
 - Una volta che un nodo è stato estratto dalla coda di priorità non verrà più reinserito.
- Le operazioni $insert()$ e $decreaseKey()$ hanno costo $O(\log n)$ e sono eseguite al più m volte.
 - Una volta per ogni arco.
- Totale: $O((n+m) \log n) = O(m \log n)$ se tutti i nodi sono raggiungibili dalla sorgente.

Costo computazionale:

n = numero archi

m = numero vertici

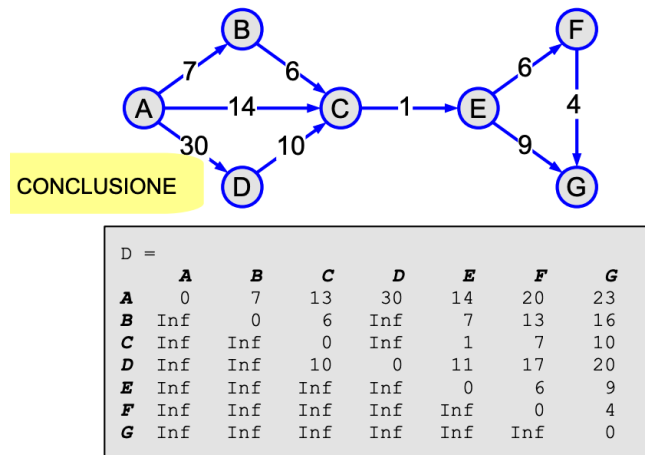
$$O(m \log n)$$

12.4 Algoritmo di Floyd e Warshall

- Si può applicare a grafi orientati con costi arbitrari (anche negativi), purché non ci siano cicli negativi
- Sia $V = \{1, 2, \dots, n\}$
- Sia D_{xy}^k la distanza minima dal nodo x al nodo y , nell'ipotesi in cui gli eventuali nodi intermedi possano appartenere esclusivamente all'insieme $\{1, \dots, k\}$
- La soluzione al nostro problema è D_{xy}^n per ogni coppia di nodi x e y .

$$D_{xy}^k = \min\{D_{xy}^{k-1}, D_{xk}^{k-1} + D_{ky}^{k-1}\}$$

Esempio:



Costo computazionale:

$$O(n^3)$$

12.5 Algoritmo di FloydWarshall

```
double[1..n,1..n] FloydWarshall2( G=(V,E,w) )
int n ← G.numNodi();
double D[1..n, 1..n];
int x, y, k, next[1..n, 1..n];
for x ← 1 to n do
  for y ← 1 to n do
    if (x == y) then D[x,y] ← 0;
    elseif ((x,y) ∈ E) then D[x,y] ← w(x,y);
    else D[x,y] ← +∞;
    endif
  endfor
endfor
for k ← 1 to n do
  for x ← 1 to n do
    for y ← 1 to n do
      if (D[x,k] + D[k,y] < D[x,y]) then
        D[x,y] ← D[x,k] + D[k,y];
      endif
    endfor
  endfor
endfor
return D;
```

12.6 Ricostruzione dei cammini

- Per ricostruire i cammini di costo minimo possiamo usare una matrice dei successori $next[x, y]$ di $n * n$ elementi.
 - $next[x, y]$ è l'indice del secondo nodo attraversato dal cammino di costo minimo che va da x a y (il primo nodo di tale cammino è x , l'ultimo è y).

Se non sono presenti pesi negativi utilizzare **Dijkstra**, altrimenti utilizzare **Bellman Ford**.

12.7 Teoria della NP-completezza

P indica la complessità polinomiale.

- Consideriamo un problema Q come una relazione:

$$Q \subseteq I \times S$$

- I è l'insieme delle istanze di ingresso.
 - S è l'insieme delle soluzioni.
- Possiamo immaginare Q come un predicato che, dato in ingresso una istanza di input $x \in I$ e una soluzione $s \in S$, restituisce:
 - 1 se $(x, s) \in Q$ (s è soluzione del problema Q sull'istanza x)
 - 0 altrimenti (s non è soluzione del problema Q sull'istanza x)
- Data una funzione $f(n)$, chiamiamo $TIME(f(n))$ (resp. $SPACE(f(n))$) l'insieme di tutti i problemi decisionali che possono essere risolti in tempo (resp. in spazio) $O(f(n))$.
- **TIME** indica un insieme di problemi

12.7.1 Classi di complessità

- La **classe P** è la classe dei problemi risolvibili in **tempo polinomiale** nella dimensione n dell'istanza di ingresso:

$$P = \cup_{c=0}^{\infty} TIME(n^c)$$

- La **classe PSPACE** è la classe dei problemi risolvibili in **spazio polinomiale** nella dimensione n dell'istanza di ingresso:

$$PSPACE = \cup_{c=0}^{\infty} SPACE(n^c)$$

- La **classe EXPTIME** è la classe dei problemi risolvibili in **tempo esponenziale** nella dimensione n dell'istanza di ingresso:

$$EXPTIME = \cup_{c=0}^{\infty} TIME(2^{n^c})$$

Se un problema è in P allora esegue una quantità di operazioni polinomiali.

- Un algoritmo che richiede tempo polinomiale riuscirà al più ad accedere ad un numero polinomiale di locazioni di memoria diverse, quindi:

$$P \subseteq PSPACE$$

- Poiché n^c locazioni di memoria possono trovarsi al più in 2^{n^c} stati diversi, si ha anche:

$$PSPACE \subseteq EXPTIME$$

- Non è noto se le inclusioni di cui sopra sono strette (non si sa se $P \subset PSPACE$ o se $PSPACE \subset EXPTIME$), ma una delle due inclusioni è stretta! (in quanto si sa che $P \subset EXPTIME$)

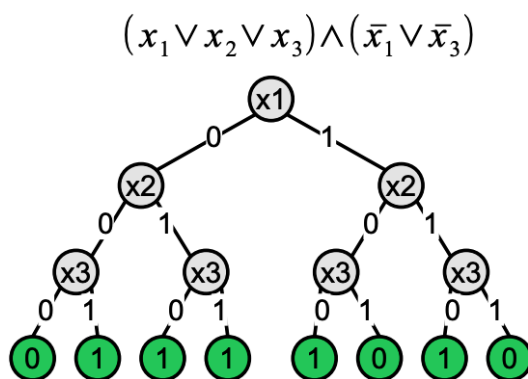
12.7.2 Verificare vs Certificare

- Come visto in precedenza, nei problemi di decisione siamo interessati a sapere se una istanza x del problema verifica una certa proprietà.
- Spesso però siamo anche interessati a conoscere un qualche oggetto y , che dipende da x e dal problema da risolvere, che possa **certificare** il fatto che x gode di tale proprietà.

Informalmente **NP** è la classe dei **problemi decisionali** che ammettono **certificati verificabili** in tempo polinomiale.

Un algoritmo decisionale **non deterministico**, invece, oltre alle normali istruzioni può eseguire istruzioni del tipo "indovina $Z \in S$ ".

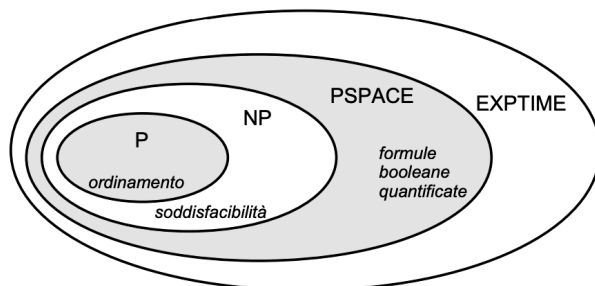
- Usando il non determinismo si può risolvere il problema della soddisfacibilità in tempo lineare.
- Nota: Un algoritmo non deterministico può essere rappresentato da un **albero di decisione**. l'algoritmo restituisce 1 se c'è almeno una foglia che restituisce 1.



La **classe NP** è la classe dei problemi risolvibili in **tempo polinomiale** non deterministico nella dimensione n dell'istanza di ingresso:

$$NP = \cup_{c=0}^{\infty} NTIME(n^c)$$

12.7.3 Gerarchia della complessità



Delle inclusioni qui sotto **almeno una** è propria:

$$P \subseteq NP \subseteq PSPACE \subseteq EXPTIME$$